

Automated Repair of TEE Partitioning Issues via DSL-Guided and LLM-Assisted Patching

CHENGYAN MA, Singapore Management University, Singapore

JIEKE SHI*, Singapore Management University, Singapore

RUIDONG HAN, Singapore Management University, Singapore

YE LIU, Singapore Management University, Singapore

FENG LI, Singapore Management University, Singapore

YUQING NIU, Singapore Management University, Singapore

DAVID LO, Singapore Management University, Singapore

Trusted Execution Environments (TEEs) provide hardware-based isolation to protect sensitive data and computations from potentially compromised operating systems (OS). However, TEE applications inevitably interact with the untrusted OS through SDK interfaces, and improper partitioning can introduce severe vulnerabilities such as data leakage and code injection. While prior work has proposed static analysis tools to detect such issues, automated repair remains largely unexplored. This problem is particularly challenging due to three TEE-specific factors: the lack of standardized secure development guidelines, the difficulty of extracting semantic information from low-level C code, and the absence of mature testing and validation methods. In this work, we present TEERepair, a framework for automatically repairing bad partitioning issues in TEE applications. Our approach tackles the above challenges by introducing a domain-specific language (DSL) to encode repair rules that express and capture common TEE security patterns, which are instantiated as patch templates with placeholders for context-specific variables. We then leverage large language models (LLMs) to reason about code semantics and synthesize context-aware patches, and further generate test clients to validate the repairs. We evaluate TEERepair on the TEE Partitioning Errors Benchmark (PartitioningE-Bench), achieving a significantly higher repair success rate of 87.6% compared to baselines. Furthermore, applying TEERepair to real-world TEE projects, we submitted 5 repair pull requests, 2 of which have been confirmed and merged by project maintainers.

CCS Concepts: • **Software and its engineering** → **Software maintenance tools**; • **Security and privacy** → **Software and application security**.

Additional Key Words and Phrases: Trusted Execution Environment, Bad Partitioning Issues, Program Repair, Domain-specific Language, Large Language Models

ACM Reference Format:

Chengyan Ma, Jieke Shi, Ruidong Han, Ye Liu, Feng Li, Yuqing Niu, and David Lo. 2026. Automated Repair of TEE Partitioning Issues via DSL-Guided and LLM-Assisted Patching. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE200 (July 2026), 24 pages. <https://doi.org/10.1145/3808207>

*Corresponding author.

Authors' Contact Information: [Chengyan Ma](#), Singapore Management University, Singapore, Singapore, chengyanma@smu.edu.sg; [Jieke Shi](#), Singapore Management University, Singapore, Singapore, jiekeshi@smu.edu.sg; [Ruidong Han](#), Singapore Management University, Singapore, Singapore, rdhan@smu.edu.sg; [Ye Liu](#), Singapore Management University, Singapore, Singapore, yeliu@smu.edu.sg; [Feng Li](#), Singapore Management University, Singapore, Singapore, fengli@smu.edu.sg; [Yuqing Niu](#), Singapore Management University, Singapore, Singapore, yuqingniu@smu.edu.sg; [David Lo](#), Singapore Management University, Singapore, Singapore, davidlo@smu.edu.sg.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2994-970X/2026/7-ARTFSE200

<https://doi.org/10.1145/3808207>

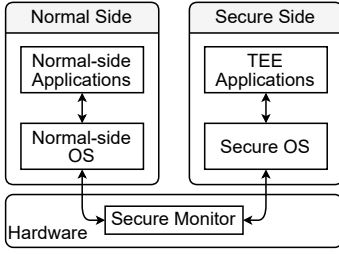


Fig. 1. System architecture of TEE.

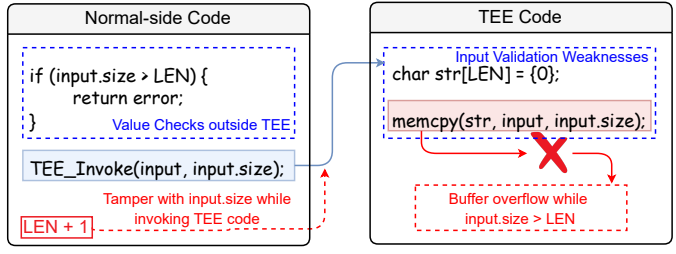


Fig. 2. An example of input validation weaknesses.

1 Introduction

Trusted Execution Environment (TEE) is an isolated area of memory and CPU that segregates sensitive code and data from the untrusted rest of the system by enforcing encryption and hardware-based isolation mechanisms (e.g., memory access control and secure CPU instructions) [11, 42]. Figure 1 illustrates the architecture of a typical TEE system such as Intel SGX [78] or ARM TrustZone [32], where programs are partitioned into two domains: the normal side, which hosts untrusted applications and the operating system, and the secure side, which contains trusted components that handle sensitive operations such as cryptography [61] or key management [49]. Such partitioning ensures only authorized code can access and manipulate sensitive data within TEE while preventing unauthorized reading or tampering by external code, thereby ensuring the confidentiality and integrity of critical computations [29, 46]. Nowadays, TEEs have been widely adopted in critical domains such as mobile payment [2], digital rights management [4], and cloud confidential computing [52], with a market projected to exceed USD 22.3 billion by 2033 [55].

Although hardware isolated, TEE applications necessarily expose software interfaces for data exchange between the normal side and TEE [12], which introduces potential weaknesses, i.e., if these interfaces are improperly designed or implemented, the isolation and confidentiality guarantees of TEE may be compromised. For example, when TEE code omits essential security mechanisms such as boundary checks, encryption, or integrity verification, attackers from the normal side can manipulate input parameters or intercept sensitive outputs. Figure 2 illustrates a typical example: while the normal-side code validates the buffer length before invoking TEE, the TEE code performs a memory copy without rechecking the size. An attacker who modifies the size parameter can trigger buffer overflows, leading to sensitive data leakage or corruption of secure execution. Similar risks occur when TEE code outputs unencrypted secrets to the normal side or directly relies on shared memory controlled by untrusted processes. These vulnerabilities are collectively referred to as bad partitioning issues [40].

Several research efforts have recognized these threats and developed tools to detect bad partitioning issues in TEE applications. In particular, Ma et al. [40] conducted a pioneering study that systematically classified bad partitioning into three categories: (1) unencrypted outputs from TEE, (2) insufficient input validation inside TEE, and (3) direct reliance on shared memory controlled by the normal side. They developed a static analyzer to detect such issues and identified hundreds of real-world vulnerabilities across open-source TEE projects, highlighting that insecure partitioning can directly lead to secret leakage, code injection, or compromise of critical services. These findings underscore that bad partitioning is not merely a theoretical concern but a prevalent and severe threat in deployed TEE applications. Yet, the problem of automatically repairing such vulnerabilities has not been systematically studied due to the following challenges:

Challenge 1: *Lack of standard TEE application development guidelines as reliable ground truths.* Although TEE SDKs [8, 17] document APIs, they do not prescribe secure partitioning practices. Unlike conventional software security with resources such as OWASP guidelines [28], TEE development lacks widely accepted standards for partitioning logic, validating inputs across trust boundaries, or managing secrets [48]. Developers often rely on ad hoc designs, resulting in inconsistent code styles and error handling [48], which prevents automated repair tools from relying on a consistent ground truth.

Challenge 2: *Hard to obtain semantic information in the low-level language.* Most TEE applications are implemented in C, a low-level language lacking memory-safety guarantees. Repairing vulnerabilities such as buffer overflows requires knowledge of semantic information, such as the intended buffer size, valid index range, or which integer field corresponds to which buffer pointer. Such information is rarely explicit in the code and is often only available through manual auditing [56]. Moreover, TEE-specific programs frequently embed these semantics in implicit ways, such as dynamically allocating memory based on an input parameter or encoding buffer lengths inside struct fields. Without recovering such semantics, repair tools risk generating incorrect patches that either truncate valid data or leave vulnerabilities unfixed.

Challenge 3: *Absence of mature testing approaches for TEE applications.* TEE applications execute in isolated environments that restrict monitoring and debugging. Intel SGX provides limited enclave debugging, while ARM TrustZone exposes results only through normal-side calls, hindering fine-grained runtime tracing. These heterogeneous mechanisms impede universal test client development [5, 9]. Our manual analysis of TEE-related GitHub projects shows most TEE applications lack unit tests or regression suites, limiting test-driven repair applicability [74]. Even when tests exist, the absence of trusted oracles (i.e., reliable reference outputs for comparison) makes it difficult to verify that patches eliminate vulnerabilities while preserving semantics. Additionally, TEE development relies on cross-compilation [59]: code is compiled with a specific toolchain on a host machine but must be deployed to TEE-enabled hardware for execution, complicating the repair-compile-test cycle.

In this paper, we propose TEERepair, a repair framework tailored for TEE applications. The core idea is to craft repair templates capturing secure programming patterns while leaving placeholders for semantic knowledge filled by large language models (LLMs). Specifically, we design a domain-specific language (DSL) that allows developers to express repair rules specifying how vulnerable code patterns should be transformed into secure alternatives. TEERepair maps detected bad partitioning issues to corresponding DSL rules and instantiates repair templates encoding the secure fix structure with placeholders for context-dependent information such as buffer sizes, variable names, or cryptographic parameters.

These placeholders are resolved by the LLM, which infers appropriate variables, constants, and operators from surrounding code context. The DSL contributes structural repair knowledge (e.g., required security checks), while the LLM supplies contextual semantics (e.g., which buffer to check, which key to use), producing syntactically valid and semantically correct patches. After each repair, we reanalyze the code using a bad partitioning detection tool [40] to ensure no new issues arise. TEERepair also automatically synthesizes TEE test clients through static analysis to verify that patches remove vulnerabilities while preserving intended functionality.

We apply TEERepair to the TEE Partitioning Errors Benchmark (PartitioningE-Bench) [40], which contains 89 cases of unencrypted data output, input validation weaknesses, and direct usage of shared memory, to assess its effectiveness in repairing bad partitioning issues. The results show that TEERepair can achieve an outstanding repair success rate of 87.6%. To further validate its practicality, we applied TEERepair to real-world TEE projects collected from GitHub. TEERepair successfully generated candidate patches for numerous identified issues, from which we submitted

5 pull requests to the corresponding repositories. At the time of writing, 2 pull requests have already been confirmed and merged by project maintainers. These results highlight the real-world applicability of TEERepair in assisting developers to secure TEE applications and demonstrate that automated repair can integrate into existing development workflows.

2 Background and Motivation

2.1 Trusted Execution Environment

A Trusted Execution Environment (TEE) provides a hardware-isolated execution context that safeguards sensitive code and data from potentially malicious software running in the untrusted Operating System (OS), often referred to as the normal side. Its security is grounded in three fundamental properties. First, isolation ensures that code and data inside TEE remain inaccessible to the normal side, even if the operating system or user-level applications are fully compromised. Second, integrity guarantees that the code and state within TEE cannot be modified or tampered with by untrusted components. Third, confidentiality prevents unauthorized disclosure of sensitive assets such as cryptographic keys or proprietary algorithms.

Technically, TEE implementations, such as ARM TrustZone and Intel SGX, combine hardware mechanisms (e.g., memory access control, secure context switching) with minimal trusted software components (e.g., a secure OS or enclave runtime) to establish a small trusted computing base (TCB) [20, 47, 59]. This design significantly reduces the attack surface compared to a full-fledged operating system. In addition, TEEs support secure boot and remote attestation [69], which enable remote parties to verify that only authorized code is loaded and executed within TEEs. Collectively, these mechanisms enhance the resilience of TEEs against privileged malware, kernel exploits, and even root-level compromises of the normal-side OS.

2.2 Bad Partitioning Issues in TEE projects

In this section, we provide a precise definition of bad partitioning issues in TEEs and explain why they pose significant threats to TEE applications. Following prior studies [38, 41, 77], we assume an adversary with full control over the normal-side execution environment, including its operating system, applications, and memory space. The adversary can arbitrarily tamper with parameters exchanged between the normal side and TEE, such as input buffers, output buffers, and shared memory regions. Specifically, the adversary can: (1) observe and modify outputs sent from TEE to the normal side, (2) craft malicious inputs or parameters before they enter TEE, and (3) alter the contents of shared memory at any time. However, the adversary cannot bypass the hardware-enforced isolation of TEE, nor can they execute arbitrary code inside it without exploiting software vulnerabilities. Under this threat model, we focus on repairing vulnerabilities that stem from bad partitioning between the normal side and TEE, including unencrypted data outputs, insufficient input validation, and unsafe reliance on shared memory.

2.2.1 Unencrypted Data Output. A common bad partitioning issue occurs when sensitive data is transferred from TEE to the normal side without encryption. Since the normal-side environment is fully under adversarial control, any plaintext data copied outside TEE can be immediately accessed by the attacker. This results in direct leakage of confidential assets such as cryptographic keys, authentication tokens, or internal state information. Even seemingly non-critical data, when leaked in plaintext—for instance through logs—can enable inference attacks [31], where adversaries analyze data variations to deduce sensitive information. To mitigate this risk, sensitive outputs should always be protected by secure mechanisms, such as encryption, before leaving TEE.

Table 1. Metadata of projects in PartitioningE-Bench.

Project	LoC	Stars	Commits	Contributors	Issues	Description
luckfox-pico	178,627	611	75	6	277	Real-world embedded SDK supporting TEE
PPFL	80,589	74	11	3	4	Privacy-preserving federated learning with TEE
darknetz	34,484	94	180	1	31	Deep Neural Network model in TEE
optee_apps	10,222	13	183	14	2	TEE applications for secure communication
optee_examples	5,380	201	99	24	57	Official OP-TEE samples
basicAlg_use	5,162	8	19	2	0	Wrapper for TEE encryption algorithm
DYMKM	1,626	147	23	3	0	An implementation of OP-TEE shared memory
Lenet5_in_OPTEE	1,610	9	32	1	1	A Lenet5 application in TEE
optee-sdp	830	3	5	1	0	Secure device storage

Table 2. Test Cases for PartitioningE-Bench, where x and y represent the data of output, input and shared memory, $\text{len}(x)$ gives the length of x , n is the bounds check value, and $x \overset{\text{NS}}{\rightsquigarrow} y$ indicates that the data x is changed to y in the normal side.

Repaired Issues	Inputs	Expected Outputs	Description
Unencrypted Data Output	—	ENC(x)	The output must be the ciphertext of x obtained by a encryption function ENC()
Input Validation Weaknesses	$\text{len}(x) < n$	TEE_SUCCESS	While an input buffer x will be copied to an in-TEE buffer, the length of x cannot be longer than the in-TEE buffer
	$\text{len}(x) = n$	TEE_SUCCESS	
	$\text{len}(x) > n$	TEE_ERROR_BAD_PARAMETERS	
	$\text{len}(x) < n$	TEE_ERROR_BAD_PARAMETERS	If an input buffer x will be access by an array index, then the length of x should be longer than the array index
	$\text{len}(x) = n$	TEE_SUCCESS	
	$\text{len}(x) > n$	TEE_SUCCESS	
Direct Usage of Shared Memory	x	TEE_SUCCESS	It should detect whether the contents of shared memory x have been tampered with to y
	$x \overset{\text{NS}}{\rightsquigarrow} y$	TEE_ERROR_BAD_PARAMETERS	

2.2.2 Input Validation Weaknesses. Another prevalent issue is insufficient validation of inputs passed from the normal side into TEE. Although TEE SDKs typically enforce basic checks (e.g., preventing null pointers), they lack semantic understanding of how inputs are used inside the secure application. Without explicit validation such as verifying buffer sizes, value ranges, or data formats, TEE code becomes vulnerable to memory safety violations, including buffer overflows and data corruption [23]. Proper input validation must therefore be implemented as a security mechanism to prevent attackers from exploiting malformed or malicious inputs.

2.2.3 Direct Usage of Shared Memory. Shared memory is a region of physical memory that can be simultaneously accessed by both the normal side and TEE to exchange data efficiently. It provides a zero-copy mechanism that avoids redundant copying of buffers across trust boundaries, making it a widely used communication channel in TEE applications. However, its direct use can compromise integrity. Since the normal side can arbitrarily modify shared memory contents at runtime, TEE code that directly reads or processes shared memory without integrity verification is exposed to tampering attacks [6, 13, 57]. To ensure correctness and security, developers should validate shared memory contents before use or adopt secure copying strategies that decouple TEE logic from untrusted memory regions.

2.3 Benchmark and Test Cases

Benchmark Construction. In this paper, we evaluate TEERepair using PartitioningE-Bench [40], a benchmark for TEE bad partitioning vulnerabilities. The benchmark construction follows a

two-phase approach. First, initial vulnerability cases were identified through manual review of real-world TEE projects collected from GitHub. Second, the dataset was systematically expanded using the CryptoAPI-Bench methodology [1], which generates comprehensive test cases by introducing variations of vulnerable patterns (such as interprocedural propagation, complex data structures, path-sensitive scenarios, and combined complexity cases) to ensure coverage of diverse vulnerability manifestations.

We acknowledge that the benchmark includes projects of varying maturity levels. To characterize project quality, we adopt the criteria proposed by Munaiah et al. [45] for identifying engineered software projects, which classifies repositories based on dimensions such as LoC, community engagement, documentation quality, commit history, and issue tracking. Table 1 summarizes the metadata of projects in the benchmark. The benchmark includes both well-maintained projects and smaller experimental ones to reflect the realistic spectrum of TEE application development. This diversity provides both ecological validity from real-world bugs and comprehensive coverage from systematically generated cases, enabling us to evaluate the effectiveness of TEERepair across varying code quality levels. In addition, the original PartitioningE-Bench contains 110 cases with both vulnerable and non-vulnerable code. To ensure dataset validity, two TEE security experts independently labeled all cases. Expert A identified 85 vulnerable cases, while Expert B identified 90. The two experts agreed on 85 cases as vulnerable and 20 cases as non-vulnerable, with a Cohen's Kappa of 0.86. After discussion with a third expert to resolve disagreements, we finalized 89 validated vulnerable cases for repair evaluation.

Test Case Construction. Since the original projects do not provide test cases, we constructed test cases for PartitioningE-Bench through a two-layer approach, ensuring 100% vulnerability coverage. For interface reachability, we statically analyze TEE source code to extract command IDs and parameter specifications from entry-point handlers, automatically generating test clients that invoke all valid TEE interfaces. This ensures that every vulnerable function entry point is reachable without manual intervention, as described in Section 3.3. For vulnerability-specific validation, we manually constructed 172 test cases (covering all 89 vulnerabilities) with inputs specifically crafted to trigger vulnerable functions and expected outputs that verify security properties after repair, as illustrated in Table 2: (1) encryption test cases verify correct ciphertext outputs, (2) input validation test cases include both valid and invalid boundary values, and (3) shared memory test cases verify deep copy and integrity check patterns. These test cases were cross-validated by two authors to ensure correctness.

3 Methodology

As shown in Fig. 3, TEERepair consists of 6 steps: ❶ We provide three rules for the repair solutions of TEE bad partitioning issues. Based on these rules, we give the template code that will be used in the following steps. ❷ Using the bad partitioning detection tool for the TEE application to get the issue report. ❸ By injecting the related information into LLM prompts and submitting the TEE source code and the corresponding repair template according to vulnerability type and code, the LLM is required to generate a repair patch. ❹ Using the bad partitioning detection tool to analyze the repaired code again to ensure that the issues are fixed, and feed the detection results back to the LLM. ❺ In order to validate the functional correctness of the repair results, we can automatically build a test client based on the repaired source code to execute test cases. ❻ We need to validate whether the output of the repaired TEE application meets the expected output of the test case to show that the repaired code does not change the original functionality of the TEE application.

As introduced in Section 2.2, prior studies have identified three types of bad partitioning issues in TEEs: unencrypted data outputs, input validation weaknesses, and direct usage of shared memory. Ma et al. [40] proposed a static analysis tool that systematically detects these issues in TEE projects.

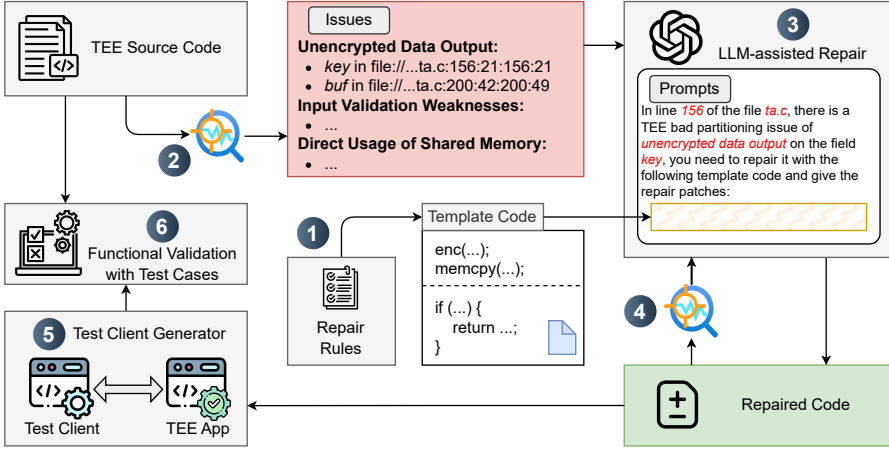


Fig. 3. Overview workflow of TEERepair. The 6 steps are: ❶ template code generating; ❷ issue locating; ❸ LLM-assisted repair; ❹ feedback; ❺ test client generating; ❻ functional validation.

In this work, we integrate their detection tool into our repair pipeline. However, detection alone is insufficient to achieve repairs, as it only locates vulnerable code fragments without providing guidance on how to fix them. To bridge this gap, our key idea is to design a domain-specific language (DSL) that allows developers and debuggers to express repair rules as structured transformations. Each DSL rule encodes insecure partitioning patterns and their secure alternatives, and can be automatically instantiated into repair templates. These templates are then combined with large language models (LLMs), which infer context-specific details such as variable names, buffer sizes, or cryptographic parameters, thereby generating semantically correct patches. The following section presents our DSL design, demonstrates how it instantiates repair templates for the three types of bad partitioning issues (Section 3.1), explains how it guides automated repair using LLMs (Section 3.2), and introduces the two-stage validation pipeline for repair patches (Section 3.3). This design systematically addresses the three challenges identified in Section 1:

- **Challenge 1 (Lack of Guidelines)**: Our DSL encodes expert knowledge about secure TEE partitioning practices into structured repair templates, serving as reliable ground truths for patch generation.
- **Challenge 2 (Missing Semantics)**: The LLM resolves context-dependent semantic information by interpreting local data-flow and natural-language cues, which traditional static analysis cannot reliably obtain.
- **Challenge 3 (No Feedback Loop)**: A two-stage strategy decouples repair from hardware constraints: static analysis provides fast feedback during repair, while generated test clients validate patches on real hardware.

3.1 DSL-Based Repair Templates

The DSL addresses **Challenge 1** by encoding established secure TEE partitioning practices into structured templates. These patterns are derived from security guidelines [8, 17, 20, 59] and real-world vulnerability analyses [40, 48], providing reliable ground truths. Fig. 4 defines the grammar of our DSL. A **Program** is a sequence of **Node** elements separated by semicolons, representing a linearized view of TEE code. Each **Node** corresponds to either a function call (**FuncCall**) or a conditional branch (**Guard**). In the Node descriptions, symbols such as `Var`, `Len`, and `Format` (line

```

1  Program      ::= Node | Node ";" Program
2  Node         ::= FuncCall | Guard
3  FuncCall     ::= COPY "(" Dest "," Src "," Len ")"
4                | SNPRINTF "(" Dest "," Format "," Args ")"
5                | MALLOC "(" Size ")" "→" Base
6                | ENC "(" Plain "," Cipher "," Len ")"
7                | MULMALLOC "(" Args ")" "→" Bases
8                | MULENC "(" Args "," Ciphers ")"
9                | ARRAY "(" Base "," Index ")" "→" Var
10               | SHALLOW "(" Base ")" "→" Var
11               | READ "(" ")" "→" Var
12               | WRITE "(" Var ")"
13               | HASH "(" Base "," Var "," Len ")"
14               | MUTATE "(" Var ")" "→" Base
15 Guard        ::= IF "(" LeftVar "," RelOp "," RightVar ")" "→" Action
16
17 Var/Len/Format/Size/Index/LeftVar/RightVar  ::= identifiers | literals
18 Base(s)/Dest/Src/Plain/Cipher(s)           ::= identifiers.address
19
20 Args     ::= Arg | Arg "," Args
21 Arg      ::= Var
22 RelOp    ::= "==" | "!=" | "<" | "<=" | ">" | ">="
23 Action   ::= return ECODE
24
25 MULMALLOC ::= MALLOC | MALLOC ";" MULMALLOC
26 MULENC    ::= ENC | ENC ";" MULENC

```

Fig. 4. Domain-Specific Language Grammar.

15 of Fig. 4) typically denote variable names, numeric values, or constant strings. Symbols in line 16 (e.g., Base and Dest) represent aliases of memory buffer base addresses, i.e., they conceptually refer to the start address of a memory region.

A **FuncCall** abstracts common memory and computation operations in TEE applications. For instance, COPY copies data from the source buffer (Src) to the destination buffer (Dest), optionally with a specified length (Len). Similarly, SNPRINTF copies data to the destination buffer but supports variable arguments (Args), which are formatted into a string according to Format. MALLOC allocates a buffer (Base) of the given Size inside TEE, while ENC performs symmetric encryption of a plaintext buffer (Plain) into a ciphertext buffer (Cipher). For multiple arguments, MULMALLOC and MULENC apply MALLOC and ENC to each argument (Arg) sequentially.

The grammar also models array and memory operations. ARRAY(Base, Index)→Var denotes accessing element Var at a specific Index of array Base, while SHALLOW(Base)→Var creates a shallow copy of a memory space. Secure storage interactions are captured by READ()→Var and WRITE(Var), representing reading from and writing to secure objects, which are TEE-protected areas for persistent data such as keys or hash values. Additionally, HASH(Base, Var, Len) computes a hash value Var over the input buffer (Base), optionally constrained by length (Len). Finally, MUTATE(Var)→Base models in-place transformations, where the value of Var is modified and stored back to Base.

A **Guard** models conditional execution with IF(LeftVar, RelOp, RightVar)→Action, where a relational operator (RelOp) compares two variables (LeftVar and RightVar) to determine if the guarded Action should be executed. RelOp may be any standard comparison operator, including equality, inequality, and relational comparisons (e.g., <, <=). The Action is expressed as return ECODE, where ECODE denotes an error or status code returned when the guard condition holds.

Overall, this grammar captures the essential semantics of TEE function calls and conditional control flow while abstracting away low-level syntax. It provides a structured and analyzable representation that enables us to systematically encode insecure programming patterns and their

(a) Rule 1.1 (DSL)	(b) Rule 1.1 (Patch Template)
<pre> 1 COPY(\$out, \$plain, \$len) → _ 2 => 3 MALLOC(\$len) -> \$cipher; 4 ENC(\$plain, \$cipher, \$len) → _; 5 COPY(\$out, \$cipher, \$len) → _ </pre>	<pre> 1 + char \$cipher[128] = {0}; 2 + enc(plain, \$cipher, 128); 3 - TEE_MemMove(params[0].memref.buffer, plain, 128); 4 + TEE_MemMove(params[0].memref.buffer, \$cipher, 128); </pre>
(c) Rule 1.2 (DSL)	(d) Rule 1.2 (Patch Template)
<pre> 1 SNPRINTF(\$out, \$format, \$args) → _ 2 => 3 MULMALLOC(\$args) → \$ciphers; 4 MULENC(\$args, \$ciphers) → _; 5 SNPRINTF(\$out, \$format, \$ciphers) → _ </pre>	<pre> 1 + char \$cipher0[strlen(arg0)] = {0}; 2 + char \$cipher1[strlen(arg1)] = {0}; 3 + enc(arg0, \$cipher0, strlen(arg0)); 4 + enc(arg1, \$cipher1, strlen(arg1)); 5 - snprintf(params[0].memref.buffer, params[0].memref. 6 size, "%s-%s", arg0, arg1); 7 + snprintf(params[0].memref.buffer, params[0].memref. 8 size, "%s-%s", \$cipher0, \$cipher1); </pre>

Fig. 5. Rule 1: Unencrypted data output repaired by adding encryption.

secure counterparts. In particular, the DSL serves as the foundation for defining repair rules, instantiating them into patch templates, and guiding the LLM to generate context-aware fixes that are both syntactically valid and semantically correct.

With the above grammar, we can formally encode bad partitioning patterns and their corresponding secure transformations. Figs. 5 to 7 illustrate three representative repair rules, each consisting of two parts: a DSL-described rule and the corresponding patch template generated from it. A rule is divided into two segments by the symbol $\Rightarrow$. The upper part denotes the vulnerable code pattern identified by the detection tool, which we call the trigger. The lower part specifies the secure replacement, referred to as the transformer. With the defined rules, our DSL will run to generate concrete patch templates by aligning the repair transformations with the actual source code. In these templates, deleted lines are prefixed with $-$, while newly added lines are prefixed with $+$. The original lines affected by the bad partitioning issue are highlighted, corresponding to the triggers defined in the DSL. By matching fields in the vulnerable code with those in the DSL trigger, most parts of the repaired TEE code can be automatically instantiated from the transformer. For fields that cannot be directly inferred (e.g., buffer sizes or variable names), we retain their symbolic representation in the DSL using a placeholder prefix ($\$$), which will subsequently be resolved by the LLM.

Repair Rule 1: Unencrypted data output repaired by adding encryption. This rule enforces encryption before sensitive data leaves TEE. Rule 1.1 (Fig. 5a) triggers when sensitive data `plain` is directly copied to output parameter `out`. The transformer replaces `COPY(out, plain, len)` with `COPY(out, cipher, len)` by: 1) declaring buffer `cipher` for ciphertext; 2) applying symmetric encryption `ENC` with matching size; 3) copying `cipher` to the output. Fig. 5b shows the generated template. From line 3 (to be deleted), we extract `out (params[0].memref.buffer)`, `plain (plain)`, and `len (128)`. The template declares a 128-byte buffer `$cipher` (line 1), encrypts via `enc()` (line 2), and copies `$cipher` to the output (line 4). For variadic functions like `snprintf` that output multiple data, Rule 1.2 (Fig. 5c, lines 3 to 4) generates corresponding ciphertext for each argument. The resulting template (Fig. 5d) expands `MULMALLOC` and `MULENC` into multiple memory initialization and encryption statements.

Repair Rule 2: Input validation weaknesses repaired by inserting a guard. Unvalidated inputs into the TEE may cause memory corruption vulnerabilities such as buffer overflow. To fix this, we generate validation code placed before input usage. Rule 2 (Fig. 6) handles two triggers: copying

(a) Rule 2.1 (DSL)	(b) Rule 2.1 (Patch Template)
<pre> 1 COPY(\$dst, \$in, \$len) → _ 2 => 3 IF(\$len, >, \$value) → return ECODE; 4 COPY(\$dst, \$in, \$len) → _ </pre>	<pre> 1 + if (params[1].memref.size > \$value) { 2 + return TEE_ERROR_BAD_PARAMETERS; 3 + } 4 + TEE_MemMove(buf, params[1].memref.buffer, 5 + params[1].memref.size); </pre>
(c) Rule 2.2 (DSL)	(d) Rule 2.2 (Patch Template)
<pre> 1 ARRAY(\$base, \$index) → _ 2 => 3 IF(\$index, >, \$value) → return ECODE; 4 IF(\$index, <, 0) → return ECODE; 5 ARRAY(\$base, \$index) → _ </pre>	<pre> 1 + if (params[0].value.a - 8 > \$value) { 2 + return TEE_ERROR_BAD_PARAMETERS; 3 + } 4 + if (params[0].value.a - 8 < 0) { 5 + return TEE_ERROR_BAD_PARAMETERS; 6 + } 7 + array[params[0].value.a - 8] = 43; </pre>

Fig. 6. Rule 2: Input validation weaknesses repaired by inserting a guard.

(a) Rule 3.1 (DSL)	(b) Rule 3.1 (Patch Template)
<pre> 1 SHALLOW(\$sm) → \$buf 2 => 3 MALLOC(\$size) → \$buf; 4 COPY(\$buf, \$sm, \$size) → _; 5 READ() → \$h1; 6 HASH(\$buf, \$h2, \$size) → _; 7 IF(equal(\$h1,\$h2), !=, 0) → return ECODE </pre>	<pre> 1 - char* buf = params[3].memref.buffer; 2 + char buf[\$size] = {0}; 3 + TEE_MemMove(buf, params[3].memref.buffer, \$size); 4 + char \$hash1[256]; 5 + read(\$hash1); 6 + char \$hash2[256]; 7 + hash(\$hash2, buf, \$size); 8 + if (TEE_MemCompare(\$hash1, \$hash2, 256) != 0) { 9 + return TEE_ERROR_BAD_PARAMETERS; 10 + } </pre>
(c) Rule 3.2 (DSL)	(d) Rule 3.2 (Patch Template)
<pre> 1 MUTATE(\$value) → \$sm 2 => 3 MUTATE(\$value) → buf; 4 HASH(\$buf, \$h, \$size) → _; 5 WRITE(\$h) → _; 6 COPY(\$sm, \$buf, \$size) → _ </pre>	<pre> 1 - *(params[3].memref.buffer + 10) = 0x55; 2 + *(\$buf + 10) = 0x55; 3 + char \$hash[256]; 4 + hash(\$hash, \$buf, \$size); 5 + write(\$hash); 6 + TEE_MemMove(params[3].memref.buffer, \$buf, \$size); </pre>

Fig. 7. Rule 3: Direct usage of shared memory repaired by deep copy and integrity check.

input parameter `in` to in-TEE buffer `dst` (Rule 2.1), and accessing array base with an index (Rule 2.2), where the input can be either the array itself or an access index. The transformer constructs conditional statements comparing sizes or checking index values accordingly. In Rule 2.1, input length must not exceed the in-TEE buffer (line 3 of Fig. 6a); in Rule 2.2, the index must be within valid bounds (lines 3 to 4 of Fig. 6c). The generated templates are shown in Fig. 6b and Fig. 6d. Fig. 6b (lines 1 to 3) adds a value check for `params[1].memref.size`, returning `TEE_ERROR_BAD_PARAMETERS` if invalid. Similarly, Fig. 6d includes bounds checks to prevent array out-of-bounds access.

Repair Rule 3: Direct usage of shared memory repaired by deep copy and integrity check. Rule 3 (Fig. 7) replaces shallow copies of shared memory with deep copies and adds integrity verification. Rule 3.1 triggers on shallow copy statements (line 1 of Fig. 7a) and constructs a buffer `buf` with matching name and size to deep copy data into the TEE. Following existing work [38, 41], we verify integrity by comparing the original hash `h1` from a secure object with the recalculated hash `h2` of the copied data. If the two are inconsistent, it means that the shared memory data has

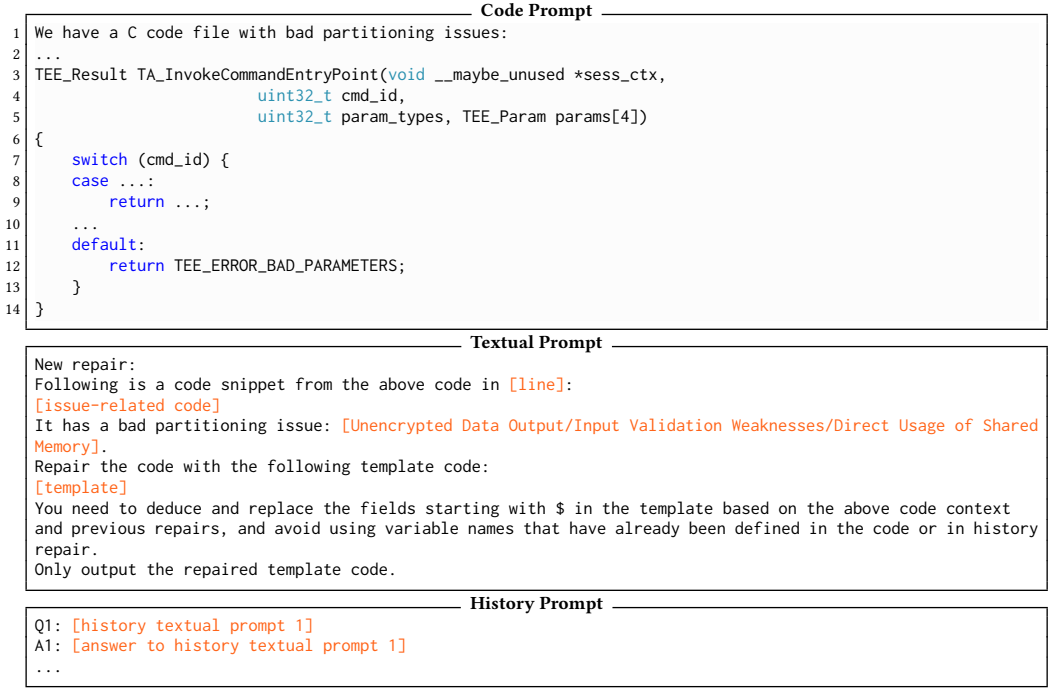


Fig. 8. Prompt for generating the repair patch.

been modified. The template in Fig. 7b performs deep copy (line 3) to replace shallow copies (e.g., line 1), with integrity checks in lines 4 to 10. For operations modifying shared memory (line 1 of Fig. 7d), Rule 3.2 ensures changes are first performed on the deep copy buf, then copied back after updating the secure object. As shown in Fig. 7d, operations on `params[3].memref.buffer` are redirected to `buf` (lines 1 to 2), and lines 3 to 6 update the integrity value and transfer data back to shared memory.

3.2 LLM-Based Placeholder Synthesis

Most TEE projects are developed using native languages such as C or C++. However, static analysis tools for C/C++ often struggle with accurately analyzing cases such as dynamically allocated memory sizes and complex pointer behaviors (e.g., `void *buffer` lacks explicit associations with their semantic constraints, such as corresponding length or index variables). Moreover, due to the strong isolation properties of TEE, applying dynamic analysis to extract runtime stack or heap-related information is also highly constrained or even infeasible. However, the above information is usually needed for repairs, such as patches for buffer size checks or memory copy operations (**Challenge 2**). To address these limitations, this section leverages the code understanding capabilities of LLMs to assist in generating repair patches targeted at bad partitioning issues in TEE projects.

As shown in Fig. 8, we give the prompts to generate repair patches for bad partitioning issues. The textual prompt includes the line number, issue name, issue-related code, and other information that can help the LLM understand the issue. Then, we also provide the template code defined in Section 3.1. Finally, the LLM is asked to fill in the placeholder fields starting with \$ in the template as the repair patch for the corresponding issue. In addition, to ensure that LLM-generated code

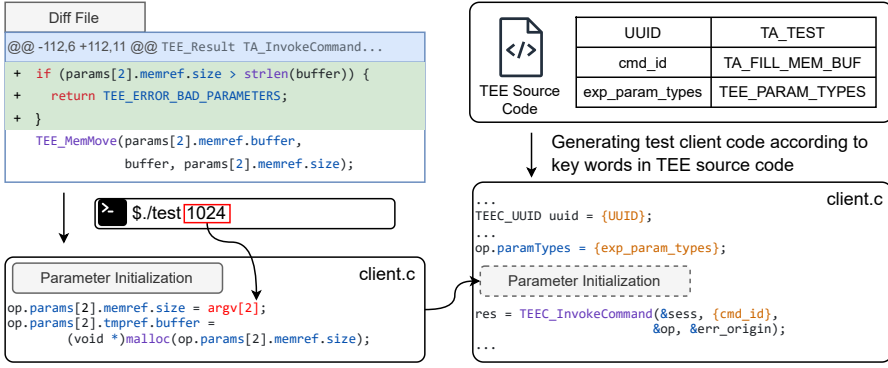


Fig. 9. Workflow of generating the test client for a TEE application.

does not have the problem of repeatedly defining variable names after filling the fields of the template, each prompt we upload will also include the complete TEE code and historical questions and answers. Since the LLM is not required to generate code and only needs to fill in the key parts of the template code, the uncertainty when the LLM repairs the code can be avoided.

3.3 Two-Stage Validation Pipeline

Due to the unreliability of LLM-generated results, it is necessary to validate the correctness of any repairs through test cases. These test cases should be used to verify not only whether the repair resolves the TEE bad partitioning issue, but also whether it preserves the original functionality. However, testing TEE applications presents some challenges. First, each TEE application typically accepts a specific set of parameters, and the accepted parameter types and structures vary across different applications. Moreover, the normal-side client programs responsible for interacting with the TEE applications are often tightly coupled with the application functions and include large amounts of code unrelated to testing, making them unsuitable for use as test clients. In addition, some TEE projects do not even provide client programs. As a result, there is no client that can be used for testing TEE applications in a general-purpose framework. What's more, TEE development requires cross-compilation toolchains and secure hardware deployment, making frequent generate-compile-test iterations quite expensive for each attempt (**Challenge 3**). TEERepair adopts a two-stage strategy to decouple repair from hardware constraints:

Stage 1: Repair Phase (Static Feedback). The static analyzer DITING [40] acts as a fast proxy oracle. After the LLM generates a candidate patch, DITING re-analyzes the patched code to check whether vulnerable patterns have been eliminated. If vulnerabilities persist, feedback is provided to the LLM for refinement. This iterative process operates entirely on source code without requiring hardware execution, enabling rapid patch iteration.

Stage 2: Validation Phase (Runtime Signals). Once static analysis confirms the absence of vulnerable patterns, patches are deployed to real TEE hardware for functional validation. TEERepair automatically generates test clients that invoke the patched TEE Application through standard TEE interfaces. This phase verifies that patches (1) compile without errors under the target TEE SDK, (2) execute successfully on secure hardware, and (3) produce expected outputs that match the intended security semantics (e.g., encrypted data for sensitive outputs, proper validation results for untrusted inputs and shared memory). Any failures observed are leveraged to refine DSL rules for future repairs.

Fig. 9 shows the workflow for generating test clients and test cases used to validate TEE repairs. The process begins with analyzing the diff file to identify changes introduced in the patched code. For instance, a newly added condition may enforce that the memory size of the input parameter `params[2].memref` must not exceed the length of a buffer, as shown in the diff segment. Based on this constraint, we need to provide test cases, such as specific values passed via command-line arguments (e.g., 1024) for parameter initialization to trigger the repaired boundary condition.

Meanwhile, metadata (e.g., the application UUID, command ID, and parameter type) is extracted from the TEE source code. This information is used to automatically construct a test client (`client.c`), which encapsulates only the minimal logic required to invoke the trusted function under test. The client sets up parameters based on user input, allocates memory accordingly, and calls `TEEC_InvokeCommand()` with related metadata.

Finally, with the test cases in Section 2.3, we can validate the functional correctness of the repaired code. For example, for the issues of unencrypted data output, we should confirm whether the TEE application returns the expected output which is correctly encrypted. For input validation weaknesses, the invalid or boundary-triggering inputs (e.g., excessively large memory sizes) must be rejected by the patched code, and the TEE application should return the appropriate error code, such as `TEE_ERROR_BAD_PARAMETERS`.

4 Evaluations

To evaluate the effectiveness of our proposed tool, we have conducted several experiments to answer the following research questions:

- RQ1.** How accurate is TEERepair in targeting issue locations compared to the basic LLM?
- RQ2.** How effective is TEERepair at repairing the TEE bad partitioning issues on the benchmark?
- RQ3.** How do the individual components of TEERepair contribute to its repair effectiveness?
- RQ4.** How does TEERepair perform on real-world TEE projects?

4.1 Environmental Setups

We implemented the transformation from DSL rules to patch templates in Python. To support the generation of repair patches for TEE applications, we leveraged the `gpt-4.1-mini-2025-04-14` model. In addition, we combined CodeQL [16], a widely used semantic code analysis engine that allows writing queries over program abstract syntax trees and control/data-flow graphs, with Python scripts to extract semantic information from TEE applications and to automatically generate corresponding test clients. All patch generation and test client creation were performed on a server running Ubuntu 24.04, equipped with a 48-core 2.3 GHz AMD EPYC 7643 processor and 512 GB of RAM. The compiled TEE applications and test clients were then deployed and executed on a Raspberry Pi 3B board with a hardware-backed TEE. Finally, the functional correctness of the generated patches was evaluated against a comprehensive suite of test cases.

Baseline Selection. For comparison, we employed a conversational GPT system with question-answer interactions as a baseline, following the design of the prior LLM-based APR tool, ChatRepair [66, 67]. We also compare with representative APR tools, including TBar [35] and Prophet [37] (template-based), and AVR tools like VulMaster[79] (learning-based), SAN2PATCH [25] (LLM-based). These tools were selected because their repair processes do not rely entirely on code compilation and execution, thereby avoiding the challenges of TEE cross-compilation and hardware deployment. Since these tools were not designed for TEE, we made adaptations: (1) TBar, Prophet, and VulMaster require precise vulnerability localization, so we provided them with exact line numbers of vulnerable code from DITING [40]; we also extended TBar and Prophet with DSL-defined templates of TEERepair while preserving their original patch generation algorithms; for VulMaster,

Table 3. The precision and recall of issue locating by TEERepair and the LLM on PartitioningE-Bench. NI is the number of partitioning issues in the benchmark, N is the number of results that tools report, and TP is the number of true positives. P, R and F1 indicates the precision, recall, and F1 score, respectively*.

Bad Partitioning Issues	NI	TEERepair					LLM-based Localization				
		N	TP	P(%)	R(%)	F1	N	TP	P(%)	R(%)	F1
Unencrypted Data Output	35	33	33	100	94.29	0.97	35	35	100	100	1
Input Validation Weaknesses	29	23	23	100	79.31	0.88	19	19	100	66.52	0.79
Direct Usage of Shared Memory	25	29	25	86.21	100	0.93	5	0	0	0	0
Total	89	85	81	95.29	91.01	0.93	59	54	91.53	60.67	0.73

* Note: $P(\%) = TP/N \times 100$, $R(\%) = TP/NI \times 100$.

```

1 void copy(char *str, int size)
2 {
3     TEE_MemMove(str, data, strlen(data));
4 }
5 ...
6 copy(params[0].memref.buffer,
   ↪  params[0].memref.size);

```

Fig. 10. Undetected issues of unencrypted data output with TEERepair.

```

1 int a[15] = {...};
2 int str[15] = {...};
3
4 int v = a[params[1].value.a];
5
6 char *array = (char *)params[2].memref.buffer;
7 char c = array[15];

```

Fig. 11. Undetected issues of input validation weaknesses with the LLM.

we used its original pre-trained model due to the lack of TEE-specific training data (Challenge 1); (2) ChatRepair relies on its LLMs to infer vulnerable locations; for SAN2PATCH, we replaced sanitizer reports with static analysis reports from DITING [40] for vulnerability localization, as sanitizers cannot be deployed in TEEs due to hardware isolation. The limitations of existing AVR tools for TEE repair are discussed in Section 6.2.

4.2 RQ1: Accuracy of Targeting Issue Locations

Since localization techniques remain inadequate in the existing tools [18], we evaluate TEERepair and a LLM-based vulnerability localization method on PartitioningE-Bench, which contains 89 bad partitioning issues. As shown in Table 3, TEERepair achieves superior performance with 95.29% precision and 91.01% recall ($F1 = 0.93$), significantly outperforming the LLM (91.53% precision, 60.67% recall, $F1 = 0.73$). The only exception is unencrypted data outputs, where the LLM attains perfect precision and recall. However, this advantage is limited to a narrow parameter-passing pattern: as shown in Fig. 10, when parameters are passed as raw pointers (copy), static analysis loses contextual information, whereas the reasoning ability helps the LLM in this specific case.

For input validation weaknesses, TEERepair achieves 100% precision with 79.31% recall, matching or exceeding the LLM (100% and 66.52%). For direct usage of shared memory, TEERepair detects all cases (recall = 100%, $F1 = 0.93$), while the LLM fails entirely (recall = 0%). The LLM misses cases for two reasons. First, it only considers inputs serving as array indexes (line 4 of Fig. 11), overlooking cases where the input itself is the accessed array (line 7). In the latter case, the buffer size parameter `params[2].memref.size` does not appear explicitly, causing detection failures. TEERepair examines all array access indexes and captures these cases. Second, LLMs lack domain knowledge about TEE shared memory, which is declared in normal-world code rather than inside TEE. Five issues reported by the LLM correspond to none of the actual shared memory accesses, whereas TEERepair correctly flags all such statements.

Answer to RQ1: TEERepair is significantly more accurate at targeting issue locations than the LLM. TEERepair achieves an overall precision of 95.29% and recall of 91.01% ($F1 = 0.93$), whereas the LLM reaches only 91.53% precision and 60.67% recall ($F1 = 0.73$).

4.3 RQ2: Correctness of Repair Patches

To evaluate repair correctness, we run the test cases from Section 2.3 on PartitioningE-Bench using the test client generated in Section 3.3. A patch is considered plausible if it compiles and passes all tests. We then manually verify whether each plausible patch addresses the root cause of the vulnerability according to the security semantics defined in our DSL rules. Table 4 reports the results on 89 issues. Notably, plausible and correct patch counts are closely aligned across tools. None of the evaluated tools use test feedback to guide patch generation; instead, each relies solely on its own strategy (DSL rules, templates, or learned patterns). Test cases serve to verify whether patches faithfully execute the required security operations (e.g., specific ciphertext outputs, precise boundary conditions, copy-then-use for shared memory). Since these behaviors are deterministic, passing all tests generally indicates correct security semantics.

For Unencrypted Data Output, TEERepair correctly repairs 33/35 issues (94.3%), significantly outperforming all baselines. Although we extended TBar and Prophet with DSL-defined encryption templates, they rarely associate TEE vulnerable code with the correct vulnerability type (only 6 successes). VulMaster, lacking TEE-specific training data, fails to generate any correct patches, confirming the limitation of learning-based approaches on emerging vulnerabilities (Challenge 1). SAN2PATCH, designed for memory corruption, relies on crash reports for localization, but TEE encryption vulnerabilities do not crash, leaving SAN2PATCH unable to understand or localize these security issues. ChatRepair produces 11 plausible patches, yet none are semantically correct. As shown in Fig. 12, it misuses a non-standard in-place encryption function (lines 1-6) from the benchmark codebase, so the second call (line 11) re-encrypts the ciphertext of the first call (line 8), which then cannot be decrypted. The template code of TEERepair avoids this by allocating fresh memory for each ciphertext.

For Input Validation Weaknesses, TEERepair produces 20 correct patches (67.0%), outperforming ChatRepair (12, 41.4%), TBar (11, 37.9%), Prophet (6, 20.7%), SAN2PATCH (5, 17.2%), and VulMaster (1, 0.03%). TBar and Prophet achieve better template selection here since boundary checking is a common pattern, but struggle with instantiation due to language-specific differences. TBar, originally for Java, cannot handle C/C++ buffer semantics. It cannot use `sizeof()` or other pointer arithmetic to determine buffer lengths, which Java achieves with the `length()` method. For example, when repairing a buffer overflow vulnerability with a dynamically allocated buffer declared as `void *buffer`, it generates `if (len <= sizeof(buffer))`, which only obtains the pointer size (typically 4 or 8 bytes) rather than the actual allocated buffer size. Prophet targets C/C++ and handles static arrays but also fails on dynamic buffers. TEERepair instead leverages LLM-based placeholder synthesis to infer such semantics accurately. ChatRepair and SAN2PATCH can generate correct validation conditions but lack precise localization without guidance from DITING.

For Direct Usage of Shared Memory, TEERepair is the only tool that successfully generates correct patches. All baselines fail completely in this category because it requires understanding the semantics of shared memory. TEERepair leverages DITING to precisely localize code locations where shared memory is misused, then applies a template with LLM-based synthesis to generate correct patches. For example, most baselines insert `if (shared_mem != NULL)` before the vulnerable code, treating it as a null-pointer issue, whereas TEERepair generates `memcpy(in_TEE_buf, shared_mem, len)` followed by operations on `in_TEE_buf`, correctly enforcing the TEE security model for shared memory.

Table 4. Patch generation results* of TEERepair and baselines on PartitioningE-Bench.

Tools	Category	Unencrypted Data Output	Input Validation Weaknesses	Direct Usage of Shared Memory	Total
TEERepair	DSL + LLM	33 / 33 / 33 (94.3%) (94.3%)	23 / 20 / 20 (67.0%) (67.0%)	25 / 25 / 25 (100%) (100%)	81 / 78 / 78 (87.6%) (87.6%)
TBar	Template-based	33 / 6 / 6 (17.1%) (17.1%)	23 / 11 / 11 (37.9%) (37.9%)	25 / 0 / 0 (0.0%) (0.0%)	81 / 17 / 17 (19.1%) (19.1%)
Prophet	Template-based	33 / 0 / 0 (0.0%) (0.0%)	23 / 6 / 6 (20.7%) (20.7%)	25 / 0 / 0 (0.0%) (0.0%)	32 / 6 / 6 (0.07%) (0.07%)
VulMaster	Learning-based	33 / 0 / 0 (0.0%) (0.0%)	23 / 1 / 1 (0.03%) (0.03%)	25 / 0 / 0 (0.0%) (0.0%)	81 / 1 / 1 (0.01%) (0.01%)
ChatRepair	LLM-based	35 / 11 / 0 (31.4%) (0.0%)	19 / 12 / 12 (41.4%) (41.4%)	0 / 0 / 0 (0.0%) (0.0%)	54 / 23 / 12 (25.8%) (13.4%)
SAN2PATCH	LLM-based	2 / 0 / 0 (0.0%) (0.0%)	5 / 5 / 5 (17.2%) (17.2%)	3 / 0 / 0 (0.0%) (0.0%)	10 / 5 / 5 (0.06%) (0.06%)

Note: PartitioningE-Bench contains 89 issues: 35 unencrypted data output, 29 input validation weaknesses, and 25 direct usage of shared memory.

* The numbers are presented in the form of #Generated patches / #Plausible patches / #Correct patches.

```

1 void enc(char *str, int len)
2 {
3     for (int i = 0; i < len; i++) {
4         str[i] = str[i] + 1;
5     }
6 }
7 ...
8 + enc(key, strlen(key));
9   TEE_MemMove(params[0].memref.buffer, key, strlen(key));
10
11 + enc(key, strlen(key));
12   TEE_MemMove(params[1].memref.buffer, key, strlen(key));

```

Fig. 12. Overfitting repair on the issues of unencrypted data output by ChatRepair.

Answer to RQ2: TEERepair achieves the highest repair success rate, correctly fixing 78 out of 89 vulnerabilities (87.6%), significantly outperforming all baselines. TBar, the second-best performer, correctly repairs 17 vulnerabilities (19.1%), followed by ChatRepair (12, 13.4%), Prophet (6, 0.07%), SAN2PATCH (5, 0.06%), and VulMaster (1, 0.01%).

4.4 RQ3: The Ablation Study

To evaluate the contribution of each component, we designed two variants: DSL-only and LLM-only. Table 5 shows the results that TEERepair achieves 78 correct patches (87.6%), significantly outperforming both DSL-only (20, 22.5%) and LLM-only (12, 13.4%). This demonstrates that neither component alone is sufficient, and their combination is essential for effective TEE vulnerability repair.

The DSL-only approach already obtains vulnerability types and locations from DITING, eliminating the template selection challenge faced by TBar and Prophet. However, its primary limitation lies in placeholder synthesis. Without LLM assistance, DSL-only cannot infer which specific variables in TEE code correspond to buffer sizes, pointers, or shared memory regions. This is particularly challenging due to diverse naming conventions and complex parameter structures in the TEE code (e.g., `params[x].memref.size`).

The LLM-only approach, like ChatRepair, struggles with vulnerability localization. Without guidance from DITING and structured templates, the LLM often fails to identify the correct code

Table 5. Patch generation results* of the ablation study.

	Unencrypted Data Output	Input Validation Weaknesses	Direct Usage of Shared Memory	Total
TEERepair	33 / 33 / 33 (94.3%) (94.3%)	23 / 20 / 20 (67.0%) (67.0%)	25 / 25 / 25 (100%) (100%)	81 / 78 / 78 (87.6%) (87.6%)
DSL-only	33 / 9 / 9 (25.7%) (25.7%)	23 / 11 / 11 (37.9%) (37.9%)	25 / 0 / 0 (0.0%) (0.0%)	81 / 20 / 20 (22.5%) (22.5%)
LLM-only	35 / 11 / 0 (31.4%) (0.0%)	19 / 12 / 12 (41.4%) (41.4%)	0 / 0 / 0 (0.0%) (0.0%)	54 / 23 / 12 (25.8%) (13.4%)

Note: PartitioningE-Bench contains 89 issues: 35 unencrypted data output, 29 input validation weaknesses, and 25 direct usage of shared memory.

* The numbers are presented in the form of #Generated patches / #Plausible patches / #Correct patches.

locations requiring modification or to use the correct functions to implement the repair. This limitation has been confirmed in our localization accuracy analysis (Section 4.2), where the LLM-based approach achieves significantly lower precision compared to DITING-guided methods.

Answer to RQ3: DSL templates encode essential domain knowledge and ensure correct API usage, while LLM synthesis resolves context-dependent placeholders that the DSL-only approach cannot handle.

4.5 RQ4: In-the-Wild Study

We applied TEERepair to the TEE projects collected on GitHub and submitted the repairs as 5 pull requests, 2 of which have been confirmed by developers and merged into the main branch. For example, the project *basicAlg_use* is an in-TEE library that includes various cryptographic algorithms such as RSA, SHA, AES, and random number generation. After detecting an issue of input validation weakness in the project that could lead to a buffer overflow, we used TEERepair to repair it and successfully generated check conditions for the input data.

Another project is *PPFL*, which is a privacy-preserving federated learning framework with TEE. The project uses a lot of code to access the input buffer in array form, but does not verify the array index and the size of the input buffer. After the repair, we generated a patch containing 15 lines of code changes that resolved the issue, and the patch was accepted by the developer.

We also have some issues that we are working on with developers. For instance, the *optee-sdp* project (an application for securely storing device data in the TEE registers) used `snprintf(dest, size, format, args)` to store log messages in a buffer (`dest`) without verifying whether the buffer size was sufficient to accommodate the entire log content. In addition, the log outputs were left unencrypted, exposing a large volume of TEE internal register data to the normal side and creating a risk of sensitive information leakage. TEERepair detected and repaired these issues, ensuring the security of log handling.

Answer to RQ4: TEERepair performs effectively on real-world TEE projects, successfully producing 2 repairs that are accepted and merged by project maintainers.

5 Discussions

5.1 LLM and Runtime Cost of TEERepair

We measured the number of input and output tokens of the LLM, as well as the execution time to repair each issue in order to evaluate the cost of TEERepair. On average, the input and output tokens that the LLM used to repair a bad partitioning issue in the PartitioningE-Bench are 16682 and 72, respectively, which correspond to the estimated cost of \$0.007 (input) and \$0.0001 (output)

under the gpt-4.1-mini-2025-04-14 model. The execution time ranges from approximately 1.01 to 5.62 seconds, and most of the time is spent waiting for the LLM response.

5.2 Patch Overfitting

Unlike test-driven repair approaches that risk generating patches satisfying only specific test cases [18], TEERepair generates patches via DSL-defined security rules derived from established secure TEE partitioning practices. In our workflow, test cases serve solely for post-hoc validation to confirm that the generated patches compile correctly and preserve program functionality, while they do not guide the patch generation process itself. This ensures that TEERepair targets the root cause of bad partitioning issues (e.g., adding encryption for sensitive outputs, inserting validation for untrusted inputs) rather than only satisfying test conditions. Furthermore, our successful evaluation on real-world projects (Section 4.5) demonstrates that patches generated by TEERepair generalize beyond our benchmark, as several patches have been accepted by upstream maintainers who independently verified their correctness.

5.3 Security and Privacy Considerations

TEE security protects data confidentiality and integrity during processing rather than code secrecy. Similar to cryptographic algorithms, public availability of TEE code can enhance security through peer review. For organizations requiring code confidentiality, TEERepair supports local LLM deployment as an alternative to cloud-based APIs.

5.4 Threats to Validity

5.4.1 External Validity. TEERepair primarily targets the repair of TEE applications based on ARM TrustZone, and currently lacks evaluation on platforms such as Intel SGX and AMD SEV. However, the DSL defined in this work is platform-agnostic and applicable to any TEE program development language. Therefore, the proposed approach remains broadly generalizable.

5.4.2 Construct Validity. Currently, the test cases used in the benchmark for TEERepair are manually written. Ideally, automated approaches such as fuzzing could be employed to test TEE applications. However, existing techniques [5] struggle to achieve full code coverage in the TEE setting. As a result, manual test case design remains necessary to ensure comprehensive and reliable evaluation. To mitigate potential biases, three experienced developers independently wrote the initial test cases and cross-checked each other's work. We therefore believe that the test cases used in this work are of high quality, and the threats to construct validity are minimal.

6 Related Work

6.1 Automated Program Repair

Traditional APR. Early APR techniques rely on heuristics, constraints, or templates rather than machine learning [19]. Genetic programming approaches such as GenProg [30], Astor [43], and SimFix [21] explore candidate patches satisfying test suites. Constraint-based methods like Nopol [68], Cardumen [44], and Dynamoth [10] derive patches from program semantics. Template-based tools including TBar [35], FixMiner [27], and Avatar [36] transform buggy code using predefined patterns. However, these approaches require substantial manual effort and depend heavily on test cases for validation, which is a critical limitation for TEE applications where test suites are often absent (Section 1).

Learning-Based APR. Deep learning introduced Neural Machine Translation (NMT)-based approaches [60, 73] that treat repair as translation from buggy to correct code. Representative tools include SequenceR [7], CoCoNut [39], DLFix [33], CURE [22], SelfAPR [70], RewardRepair [71],

and others [34, 73, 80, 81]. While more generalizable than traditional APR, these methods require large-scale training data that is scarce in the TEE domain and risk generating insecure patches in security-critical contexts.

LLM-Based APR. Recent approaches leverage LLMs pre-trained on massive code corpora [64, 65, 74]. Conversation-style tools like ChatRepair [66, 67] and ContrastRepair [26] refine patches through multi-turn interactions, while cloze-style methods such as AlphaRepair [65], FitRepair [63], and Repilot [62] fill missing code fragments. We adapt ChatRepair as our baseline due to its strong performance and reproducibility. Agent-based approaches like RepairAgent [3], AutoCodeRover [76], and SpecRover [51] show promise [50], but are still at an early stage for TEE development given the lack of test suites and cross-device deployment overhead.

6.2 Automated Vulnerability Repair

Unlike APR, which typically relies on test suites to locate and validate software defects, Automated Vulnerability Repair (AVR) typically leverages sanitizer-detected crashes [25, 72] or static analysis [75] to localize vulnerabilities. However, due to the hardware isolation of TEEs, sanitizers cannot be deployed to monitor the TEE execution, making crash reports unavailable and crash-based localization infeasible for TEE applications. More fundamentally, TEE partitioning issues such as unencrypted data output and direct usage of shared memory may not cause crashes, but they can violate security properties such as confidentiality and integrity, allowing attackers to steal or tamper with sensitive data without triggering any observable failure. In this work, we rely on the static analyzer DITING [40] to detect and localize bad partitioning issues in TEE applications. Therefore, this section focuses on analyzing four categories of patch generation methods: search-based, template-based, semantics-based, and learning-based approaches [18].

Search-based approaches, such as GenProg [30], formulate patch generation as a search problem so that code snippets are iteratively modified and recombined into different program variants to produce candidate patches. CrashRepair [54] combines crash-guided concolic execution with code mutation search, which can infer repair constraints at specific program locations and search for code mutations satisfying those constraints. However, search-based methods still rely on dynamic feedback for patch evaluation: GenProg requires test suites, while CrashRepair requires crash-inducing inputs. Neither is applicable to TEE bad partitioning issues, which lack comprehensive test suites (Challenge 3) and may not manifest as crashes.

Template-based approaches leverage pre-defined fix patterns to transform vulnerable code into secure forms. Representative tools include PAR [24] and TBar [35], which collect fix templates from human-written patches for Java programs. Prophet [37] ranks candidate patch templates for C programs by learning a patch correctness model from successful human patches. TEERepair adopts a similar idea by encoding secure TEE partitioning patterns into DSL-defined repair rules, and we have added TBar and Prophet as baselines in Section 4.3.

Semantics-based approaches leverage program analysis techniques such as symbolic execution or constraint solving to generate patches with formal guarantees. Dynamic approaches like Extract-Fix [15] and Concolic program repair [53] use KLEE-based symbolic/concolic execution to extract crash-free constraints from exploit traces. However, KLEE executes LLVM bitcode within a POSIX emulation layer [58], which is incompatible with TEE applications that depend on vendor-specific cross-compilation toolchains (e.g., ARM TrustZone SDKs) and secure hardware for execution. Static approaches like EffFix [75] employ Incorrectness Separation Logic (ISL) via the Pulse analyzer to detect and repair memory safety issues, such as null pointer dereferences and double-frees. However, TEE bad partitioning vulnerabilities violate security properties (confidentiality and integrity) rather than memory safety [40], placing them outside the scope of ISL-based analysis.

Learning-based approaches train neural models on historical vulnerability-fix pairs to generate patches. VulRepair [14] employs byte-pair encoding on a CodeT5 pre-trained language model to generate software vulnerability repairs. Similarly, VulMaster [79] utilizes the Fusion-in-Decoder framework with CodeT5, integrating AST structures and CWE expert knowledge to handle code without length limitation. More recently, LLM-based approaches have emerged: SAN2PATCH [25] uses LLMs to comprehend sanitizer logs, localize root causes of vulnerabilities in source code, and generate corresponding patches; PatchAgent [72] designs an autonomous agent that integrates fault localization, patch generation, and validation, simulating the repair workflow of human experts. However, these approaches face two key limitations in the TEE development. First, they rely on frequent generate-compile-test iterations, which become expensive due to cross-compilation and hardware deployment requirements (Challenge 3). Second, bad partitioning is an emerging vulnerability class that has only recently gained attention from the security community [40, 48], and consequently, no large-scale dataset of such vulnerabilities and their fixes exists for training learning-based models (Challenge 1).

In addition, these AVR systems assume monolithic software with direct memory access, whereas TEE applications span two isolated execution environments and communicate through constrained interfaces such as shared memory and remote procedure calls. These cross-world interactions introduce unique attack surfaces that do not appear in the above AVR settings. A recent SoK paper [18] systematically surveys AVR methods, finding that vulnerability localization accuracy critically impacts repair effectiveness, yet existing localization techniques remain inadequate, and that fuzzing-based and symbolic execution-based tools often produce overfitting patches. TEERepair addresses both: we leverage DITING [40] for precise localization of TEE bad partitioning vulnerabilities, and generate templates through DSL-defined security rules encoding expert knowledge rather than test-suite guidance, thereby ensuring root-cause fixes. LLMs then synthesize context-aware code to complete these templates, enhancing both their adaptability and scalability. This synergy between rule-based guidance and LLM-assisted synthesis offers a novel perspective on automated repair for security-critical, low-level TEE applications.

7 Conclusion and Future Work

In this paper, we presented TEERepair, a framework for automatically repairing TEE bad partitioning issues. By combining DSL-based secure programming patterns with LLM semantic reasoning, TEERepair generates context-aware patches that are both syntactically valid and semantically correct. Our evaluation on PartitioningE-Bench demonstrates a repair success rate of 87.6%, significantly outperforming baselines, and our in-the-wild study confirms practical applicability to real-world projects. Future work includes extending DSL with additional security policies and evaluating it on industrial-scale codebases to integrate automated TEE repair into the secure development lifecycle.

Data Availability

The replication package is available at: <https://github.com/CharlieMCY/TEERepair>.

Acknowledgments

This research is supported by the National Research Foundation, Singapore, and the Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme (Proposal ID: NCR25-DeSCeM-T-SMU). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the National Research Foundation, Singapore, and the Cyber Security Agency of Singapore.

References

- [1] Sharmin Afrose, Sazzadur Rahaman, and Danfeng Yao. 2019. CryptoAPI-Bench: A Comprehensive Benchmark on Java Cryptographic API Misuses. In *2019 IEEE Cybersecurity Development (SecDev)*. 49–61.
- [2] Zaheer Ahmad, Lishoy Francis, Tansir Ahmed, Christopher Lobodzinski, Dev Audsin, and Peng Jiang. 2013. Enhancing the Security of Mobile Applications by Using TEE and (U)SIM. In *2013 IEEE 10th International Conference on Ubiquitous Intelligence and Computing and 2013 IEEE 10th International Conference on Autonomic and Trusted Computing*. 575–582.
- [3] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2025. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 2188–2200.
- [4] Davide Bove. 2024. A Large-Scale Study on the Prevalence and Usage of TEE-based Features on Android. In *Proceedings of the 19th International Conference on Availability, Reliability and Security (Vienna, Austria) (ARES '24)*. Association for Computing Machinery, New York, NY, USA, Article 29, 11 pages.
- [5] Marcel Busch, Aravind Machiry, Chad Spensky, Giovanni Vigna, Christopher Kruegel, and Mathias Payer. 2023. TEEzz: Fuzzing Trusted Applications on COTS Android Devices. In *2023 IEEE Symposium on Security and Privacy (SP)*. 1204–1219.
- [6] Liheng Chen, Zheming Li, Zheyu Ma, Yuan Li, Baojian Chen, and Chao Zhang. 2024. EnclaveFuzz: Finding Vulnerabilities in SGX Applications. In *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024*. The Internet Society.
- [7] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2021. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Transactions on Software Engineering* 47, 9 (2021), 1943–1959.
- [8] Intel Corporation. 2020. Intel(R) Software Guard Extensions Developer Guide. https://community.intel.com/legacyfs/online/drupal_files/managed/33/70/intel-sgx-developer-guide.pdf
- [9] Guoyun Duan, Yuanzhi Fu, Boyang Zhang, Peiyao Deng, Jianhua Sun, Hao Chen, and Zhiwen Chen. 2023. TEEFuzzer: A fuzzing framework for trusted execution environments with heuristic seed mutation. *Future Gener. Comput. Syst.* 144, C (July 2023), 192–204.
- [10] Thomas Durieux and Martin Monperrus. 2016. DynaMoth: dynamic code synthesis for automatic program repair. In *Proceedings of the 11th International Workshop on Automation of Software Test (Austin, Texas) (AST '16)*. Association for Computing Machinery, New York, NY, USA, 85–91.
- [11] Jan-Erik Ekberg. 2016. Hardware Isolation for Trusted Execution. In *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices (Vienna, Austria) (SPSM '16)*. Association for Computing Machinery, New York, NY, USA, 1.
- [12] Yongkai Fan, Guanqun Zhao, Kuan-Ching Li, Bin Zhang, Gang Tan, Xiaofeng Sun, and Fanglue Xia. 2020. SNPL: One Scheme of Securing Nodes in IoT Perception Layer. *Sensors* 20, 4 (2020).
- [13] Fabian Fleischer, Marcel Busch, and Phillip Kuhrt. 2020. Memory corruption attacks within Android TEEs: a case study based on OP-TEE. In *Proceedings of the 15th International Conference on Availability, Reliability and Security (Virtual Event, Ireland) (ARES '20)*. Association for Computing Machinery, New York, NY, USA, Article 53, 9 pages.
- [14] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: a T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (Singapore, Singapore) (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 935–947.
- [15] Xiang Gao, Bo Wang, Gregory J. Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2021. Beyond Tests: Program Vulnerability Repair via Crash Constraint Extraction. *ACM Trans. Softw. Eng. Methodol.* 30, 2, Article 14 (Feb. 2021), 27 pages.
- [16] GitHub. 2025. CodeQL — codeql.github.com. <https://codeql.github.com>.
- [17] GlobalPlatform, Inc. 2018. TEE Internal Core API Specification Version 1.1.2.50. https://globalplatform.org/wp-content/uploads/2018/06/GPD_TEE_Internal_Core_API_Specification_v1.1.2.50_PublicReview.pdf
- [18] Yiwei Hu, Zhen Li, Kedie Shu, Shenghua Guan, Deqing Zou, Shouhuai Xu, Bin Yuan, and Hai Jin. 2025. SoK: Automated Vulnerability Repair: Methods, Tools, and Assessments. In *34th USENIX Security Symposium, USENIX Security 2025, Seattle, WA, USA, August 13-15, 2025*. Lujo Bauer and Giancarlo Pellegrino (Eds.). USENIX Association, 4421–4440.
- [19] Kai Huang, Zhengzi Xu, Su Yang, Hongyu Sun, Xuejun Li, Zheng Yan, and Yuqing Zhang. 2024. Evolving Paradigms in Automated Program Repair: Taxonomy, Challenges, and Opportunities. *ACM Comput. Surv.* 57, 2, Article 36 (Oct. 2024), 43 pages.
- [20] Huawei Technologies Co., Ltd. 2019. Huawei iTrustee V3.0 on Kirin 980 Security Target. <https://messervices.cyber.gouv.fr/visas/ANSSI-CC-2020-67-cible.pdf>
- [21] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (Amsterdam, Netherlands) (ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA,

- 298–309.
- [22] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1161–1173.
 - [23] Mustakimur Rahman Khandaker, Yueqiang Cheng, Zhi Wang, and Tao Wei. 2020. COIN Attacks: On Insecurity of Enclave Untrusted Interfaces in SGX. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (Lausanne, Switzerland) (ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 971–985.
 - [24] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*. 802–811.
 - [25] Youngjoon Kim, Sunguk Shin, Hyoungshick Kim, and Jiwon Yoon. 2025. Logs In, Patches Out: Automated Vulnerability Repair via Tree-of-Thought LLM Analysis. In *34th USENIX Security Symposium, USENIX Security 2025, Seattle, WA, USA, August 13-15, 2025*, Lujio Bauer and Giancarlo Pellegrino (Eds.). USENIX Association, 4401–4419.
 - [26] Jiaolong Kong, Xiaofei Xie, Mingfei Cheng, Shangqing Liu, Xiaoning Du, and Qi Guo. 2025. ContrastRepair: Enhancing Conversation-Based Automated Program Repair via Contrastive Test Case Pairs. *ACM Trans. Softw. Eng. Methodol.* 34, 8, Article 216 (Oct. 2025), 31 pages.
 - [27] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. FixMiner: Mining relevant fix patterns for automated program repair. *Empirical Softw. Engg.* 25, 3 (May 2020), 1980–2024.
 - [28] Shubham Kumar Lala, Akshat Kumar, and Subbulakshmi T. 2021. Secure Web development using OWASP Guidelines. In *2021 5th International Conference on Intelligent Computing and Control Systems (ICICCS)*. 323–332.
 - [29] Titouan Lazard, Johannes Götzfried, Tilo Müller, Gianni Santinelli, and Vincent Lefebvre. 2018. TEShift: Protecting Code Confidentiality by Selectively Shifting Functions into TEEs. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution (Toronto, Canada) (SysTEX '18)*. Association for Computing Machinery, New York, NY, USA, 14–19.
 - [30] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72.
 - [31] Ding Li, Ziqi Zhang, Mengyu Yao, Yifeng Cai, Yao Guo, and Xiangqun Chen. 2025. TEESlice: Protecting Sensitive Neural Network Models in Trusted Execution Environments when Attackers Have Pre-Trained Models. *ACM Trans. Softw. Eng. Methodol.* 34, 6, Article 166 (July 2025), 49 pages.
 - [32] Wenhao Li, Yubin Xia, and Haibo Chen. 2019. Research on ARM TrustZone. *GetMobile: Mobile Comp. and Comm.* 22, 3 (Jan. 2019), 17–22.
 - [33] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (Seoul, South Korea) (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 602–614.
 - [34] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2022. DEAR: a novel deep learning-based approach for automated program repair. In *Proceedings of the 44th International Conference on Software Engineering (Pittsburgh, Pennsylvania) (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 511–523.
 - [35] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (Beijing, China) (ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 31–42.
 - [36] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawende F. Bissyandé. 2019. AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 1–12.
 - [37] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (St. Petersburg, FL, USA) (POPL '16)*. Association for Computing Machinery, New York, NY, USA, 298–312.
 - [38] Di Lu, Minqiang Shi, Xindi Ma, Ximeng Liu, Rui Guo, Tianfang Zheng, Yulong Shen, Xuewen Dong, and Jianfeng Ma. 2023. Smaug: A TEE-Assisted Secured SQLite for Embedded Systems. *IEEE Transactions on Dependable and Secure Computing* 20, 5 (2023), 3617–3635.
 - [39] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (Virtual Event, USA) (ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 101–114.
 - [40] Chengyan Ma, Ruidong Han, Jieke Shi, Ye Liu, Yuqing Niu, Di Lu, Chuang Tian, Jianfeng Ma, Debin Gao, and David Lo. 2025. DITING: A Static Analyzer for Identifying Bad Partitioning Issues in TEE Applications. [arXiv:2502.15281](https://arxiv.org/abs/2502.15281) [cs.CR]
 - [41] Chengyan Ma, Di Lu, Chaoyue Lv, Ning Xi, Xiaohong Jiang, Yulong Shen, and Jianfeng Ma. 2024. BiTDB: Constructing A Built-in TEE Secure Database for Embedded Systems. *IEEE Transactions on Knowledge and Data Engineering* 36, 9

- (2024), 4472–4485.
- [42] Pieter Maene, Johannes Götzfried, Ruan de Clercq, Tilo Müller, Felix Freiling, and Ingrid Verbauwhede. 2018. Hardware-Based Trusted Computing Architectures for Isolation and Attestation. *IEEE Trans. Comput.* 67, 3 (2018), 361–374.
 - [43] Matias Martinez and Martin Monperrus. 2016. ASTOR: a program repair library for Java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis (Saarbrücken, Germany) (ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 441–444.
 - [44] Matias Martinez and Martin Monperrus. 2018. Ultra-Large Repair Search Space with Automatically Mined Templates: The Cardumen Mode of Astor. In *Search-Based Software Engineering - 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11036)*, Thelma Elita Colanzi and Phil McMinn (Eds.). Springer, 65–86.
 - [45] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. *Empirical Softw. Engg.* 22, 6 (Dec. 2017), 3219–3253.
 - [46] Tsunato Nakai, Daisuke Suzuki, and Takeshi Fujino. 2021. Towards Trained Model Confidentiality and Integrity Using Trusted Execution Environments. In *Applied Cryptography and Network Security Workshops: ACNS 2021 Satellite Workshops, AIBlock, AIHWS, AIoT, CIMSS, Cloud S&P, SCI, SecMT, and SiMLA, Kamakura, Japan, June 21–24, 2021, Proceedings* (Kamakura, Japan). Springer-Verlag, Berlin, Heidelberg, 151–168.
 - [47] Peng Ning. 2014. Samsung KNOX and Enterprise Mobile Security. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices (Scottsdale, Arizona, USA) (SPSM '14)*. Association for Computing Machinery, New York, NY, USA, 1.
 - [48] Yuqing Niu, Jieke Shi, Ruidong Han, Ye Liu, Chengyan Ma, Yunbo Lyu, and David Lo. 2026. What You Trust Is Insecure: Demystifying How Developers (Mis)Use Trusted Execution Environments in Practice. arXiv:2512.17363 [cs.SE]
 - [49] Minkyung Park, Jeongnyeo Kim, Youngho Kim, Eunsang Cho, Soobin Park, Sungmin Sohn, Minhyeok Kang, and Ted Taekyoung Kwon. 2020. An SGX-Based Key Management Framework for Data Centric Networking. *IEEE Access* 8 (2020), 45198–45210.
 - [50] Abhik Roychoudhury. 2025. Agentic AI for Software: thoughts from Software Engineering community. arXiv:2508.17343 [cs.SE]
 - [51] Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. 2025. SpecRover: Code Intent Extraction via LLMs. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 963–974.
 - [52] Gianluca Scopelliti, Christoph Baumann, and Jan Tobias Mühlberg. 2024. Understanding Trust Relationships in Cloud-Based Confidential Computing. In *2024 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. 169–176.
 - [53] Ridwan Shariffdeen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2021. Concolic program repair. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (Virtual, Canada) (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 390–405.
 - [54] Ridwan Shariffdeen, Christopher S. Timperley, Yannic Noller, Claire Le Goues, and Abhik Roychoudhury. 2025. Vulnerability Repair via Concolic Execution and Code Mutations. *ACM Trans. Softw. Eng. Methodol.* 34, 4, Article 105 (April 2025), 27 pages.
 - [55] Raksha Sharma. 2024. Trusted Execution Environment Market Research Report 2033. <https://growthmarketreports.com/report/trusted-execution-environment-market>
 - [56] Alex Shaw, Dusten Doggett, and Munawar Hafiz. 2014. Automatically Fixing C Buffer Overflows Using Program Transformations. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 124–135.
 - [57] Dongwook Shim and Dong Hoon Lee. 2021. SOTPM: Software One-Time Programmable Memory to Protect Shared Memory on ARM Trustzone. *IEEE Access* 9 (2021), 4490–4504.
 - [58] The KLEE Team. 2026. KLEE — klee-se.org. <https://klee-se.org/>.
 - [59] TrustedFirmware.org. 2023. About OP-TEE. <https://optee.readthedocs.io/en/latest/general/about.html>
 - [60] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *ACM Trans. Softw. Eng. Methodol.* 28, 4, Article 19 (Sept. 2019), 29 pages.
 - [61] Mark Vella, Christian Colombo, Robert Abela, and Peter Spacek. 2021. RV-TEE: secure cryptographic protocol execution based on runtime verification. *J. Comput. Virol. Hacking Tech.* 17, 3 (2021), 229–248.
 - [62] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (San Francisco, CA, USA) (ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 172–184.
 - [63] Chunqiu Steven Xia, Yifeng Ding, and Lingming Zhang. 2023. The Plastic Surgery Hypothesis in the Era of Large Language Models. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 522–534.

- [64] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1482–1494.
- [65] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (Singapore, Singapore) (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 959–971.
- [66] Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational Automated Program Repair. arXiv:[2301.13246](https://arxiv.org/abs/2301.13246) [cs.SE]
- [67] Chunqiu Steven Xia and Lingming Zhang. 2024. Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for \$0.42 Each using ChatGPT. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (Vienna, Austria) (ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 819–831.
- [68] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clément, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering* 43, 1 (2017), 34–55.
- [69] Takashi Yagawa, Tadanori Teruya, Kuniyasu Suzuki, and Hirotake Abe. 2024. Delegating Verification for Remote Attestation Using TEE. In *2024 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. 186–192.
- [70] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2023. SelfAPR: Self-supervised Program Repair with Test Execution Diagnostics. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (Rochester, MI, USA) (ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 92, 13 pages.
- [71] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural program repair with execution-based backpropagation. In *Proceedings of the 44th International Conference on Software Engineering (Pittsburgh, Pennsylvania) (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1506–1518.
- [72] Zheng Yu, Ziyi Guo, Yuhang Wu, Jiahao Yu, Meng Xu, Dongliang Mu, Yan Chen, and Xinyu Xing. 2025. PATCHAGENT: A Practical Program Repair Agent Mimicking Human Expertise. In *34th USENIX Security Symposium, USENIX Security 2025, Seattle, WA, USA, August 13-15, 2025*. Lujo Bauer and Giancarlo Pellegrino (Eds.). USENIX Association, 4381–4400.
- [73] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2023. A Survey of Learning-based Automated Program Repair. *ACM Trans. Softw. Eng. Methodol.* 33, 2, Article 55 (Dec. 2023), 69 pages.
- [74] Quanjun Zhang, Chunrong Fang, Yang Xie, YuXiang Ma, Weisong Sun, Yun Yang, and Zhenyu Chen. 2024. A Systematic Literature Review on Large Language Models for Automated Program Repair. arXiv:[2405.01466](https://arxiv.org/abs/2405.01466) [cs.SE]
- [75] Yuntong Zhang, Andreea Costea, Ridwan Shariffdeen, Davin McCall, and Abhik Roychoudhury. 2025. EffFix: Efficient and Effective Repair of Pointer Manipulating Programs. *ACM Trans. Softw. Eng. Methodol.* 34, 3, Article 69 (Feb. 2025), 27 pages.
- [76] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (Vienna, Austria) (ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1592–1604.
- [77] Shijun Zhao, Qianying Zhang, Yu Qin, Wei Feng, and Dengguo Feng. 2019. SecTEE: A Software-based Approach to Secure Enclave Architecture Using TEE. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (London, United Kingdom) (CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1723–1740.
- [78] Wei Zheng, Ying Wu, Xiaoxue Wu, Chen Feng, Yulei Sui, Xiapu Luo, and Yajin Zhou. 2021. A survey of Intel SGX and its applications. *Front. Comput. Sci.* 15, 3 (June 2021), 15 pages.
- [79] Xin Zhou, Kisub Kim, Bowen Xu, Donggyun Han, and David Lo. 2024. Out of Sight, Out of Mind: Better Automatic Vulnerability Repair by Broadening Input Ranges and Sources. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (Lisbon, Portugal) (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 88, 13 pages.
- [80] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (Athens, Greece) (ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 341–353.
- [81] Qihao Zhu, Zeyu Sun, Wenjie Zhang, Yingfei Xiong, and Lu Zhang. 2023. Tare: Type-Aware Neural Program Repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1443–1455.

Received 2025-09-12; accepted 2026-03-24